Co-ordination of multiple external representations during Java program debugging

Pablo Romero, Rudi Lutz, Richard Cox, and Benedict du Boulay
Human Centred Technology Group,
School of Cognitive & Computing Sciences
University of Sussex, Falmer, Brighton,
East Sussex, BN1 9QH, UK.
juanr@cogs.susx.ac.uk

Abstract

Java program debugging was investigated in computer science students who used a software debugging environment (SDE) that provided concurrently displayed, adjacent, multiple and linked representations consisting of the program code, a visualisation of the program, and its output.

The aim of this investigation was to address questions such as 'To what extent do programmers use each type of representation?', 'Are particular patterns of representation use associated with superior debugging performance?', 'Are graphical representations more helpful to Java programmers than textual ones?' and 'Are representations that highlight data structure more useful than those that highlight control-flow for Java debugging?'

A modified version of the Restricted Focus Viewer (RFV) - a visual attention tracking system - was employed to measure the degree to which each of the representations was used, and to record switches between representations. The experimental results are in agreement with research in the area that suggests that control-flow information is difficult to decode in an Object-Oriented language like Java. These results also suggest that graphical representations might be more useful than textual ones when the degree of difficulty of the debugging task poses a challenge to programmers. Additionally, the results link programming experience to switching behaviour, suggesting that although switches between the code and the visualisation are the most common ones, programming experience might promote a more balanced switching behaviour between the main representation, the code, and the secondary ones.

1. Introduction

When trying to perform a programming activity in everyday settings, programmers normally work with a variety of external representations as well as the program code. Some of these external representations are used in debugging packages, prototyping and visualisation tools in software development environments, or are included as part of internal and external documentation. Therefore, programming normally requires the co-ordination of multiple representations.

Probably the most typical case, at least for novice programmers, of co-ordination of external representations in programming, is working with debugging packages, a common example of a visualisation tool. Novice programmers often spend a large amount of their time attempting to understand the behaviour of programs when trying to discover errors in the code. To perform this task, novices normally work with both the program code and the debugger output, trying to co-ordinate and make sense of these representations. Yet very little is known about how multiple external representations are used for this kind of programming task.

In [18] it was shown that visual attention tracking methods, and more specifically a tool like the Restricted Focus Viewer (RFV)[2] can be used to investigate issues related to the *process* of co-ordinating multiple external representations in program debugging. Research of this type can offer important clues about the relationship between representation use and programming information types, the issue of sentential versus graphical representations, and debugging performance.

2. Co-ordination of multiple external representations in programming

Two important aspects to consider regarding the co-ordination of multiple representations in programming are *modality* and *perspective* [5].

2.1. Modality

The term 'modality' is used here to mean the representational forms used to present or display information, rather than in the psychological sense of sensory channel. A typical modality distinction here is between propositional and diagrammatic representations.

Thus, this first aspect refers to co-ordinating representations which are basically propositional with those that are mainly diagrammatic. It is not clear whether co-ordinating representations with the same modality type has advantages over working with mixed multiple representations or whether including a high degree of graphicality has potential benefits for performing the task [1].

Although programmers normally have to coordinate representations of different modalities, there has not been much research on this topic in the area of programming. One of the few examples is the GIL system [11], which attempts to provide reasoning-congruent visual representations in the form of control-flow diagrams to aid the generation and comprehension of LISP, a functional programming language which normally employs textual representations. In [11], it is claimed that this system is successful in teaching novices to program in this language; however, this work did not compare co-ordination of the same and different modalities.

Work in the algorithm animation area ([3]) has found advantages for the use of multiple representations of mixed modality. In [3], it was found that students might benefit from the dual coding that results from presenting a graphical visualisation of the program together with a textual explanation of it.

Other studies in the area have been concerned with issues related to the format of the output of debugging packages [12, 13]. Those studies have offered conflicting results about the co-ordination of representations of different modalities. In [13], it was found that subjects working with representations of the same and different modalities had similar performance, while in [12], it was reported that those working with different modalities showed a poorer performance than those working with the same modality. In both cases, participants worked with the program code and with the debugger's output. The debugger notations used by both of these studies were mostly textual. The only predominantly graphical debugging tool used by these studies was TPM [6]. While the performance of the participants of the former study [13] was similar for the textual debuggers and TPM, the subjects of the latter study [12] found working with TPM more difficult. One important difference between these two studies is that while the former used static representations, the latter employed a visualisation package (dynamic representations). The additional cognitive load of learning and using a multi-representational visualisation package may explain the difference in findings.

2.2. Perspective

Perspective is the aspect which refers to co-ordinating representations that highlight either the same or different programming information types. Computer programs are information structures that comprise different types of information [15], and programming notations usually highlight some of these aspects at the cost of obscuring others [8] (the *match-mismatch hypothesis*). Experienced programmers, when comprehending code, are able to develop a mental representation that comprises these different perspectives or information types, as well as rich mappings between them [14]. Some of these different information types are: function, data structure, operations, data-flow and control-flow. It is an open issue whether co-ordinating notations that highlight different information types will be more beneficial to programmers than working with those that highlight the same ones.

2.3. Java debugging

To date, there have been numerous investigations of debugging behaviour across a range of programming languages [7, 16, 19] and previous research has also examined the effect of representational mode upon program comprehension [10, 11, 12, 13].

However, these studies were performed mainly in the context of procedural or declarative computer languages. It is not clear whether the results generalise to the (currently popular) Object-Oriented paradigm. Research in program comprehension for Object-Oriented languages suggests that these kinds of language highlight function as well as static data element information whilst obscuring control-flow information [4, 20]. However, it is not clear whether novice programmers working with medium size programs find comprehending function and static data element information in Object-Oriented languages an easy task [21], specially because as program size increases, this sort of information tends to become diffuse.

Furthermore, debugging studies have tended not to employ debugging environments that are typical of those used by professional programmers (*i.e.* multi-representational software debugging environments, SDEs). Such environments typically permit the user to switch rapidly between

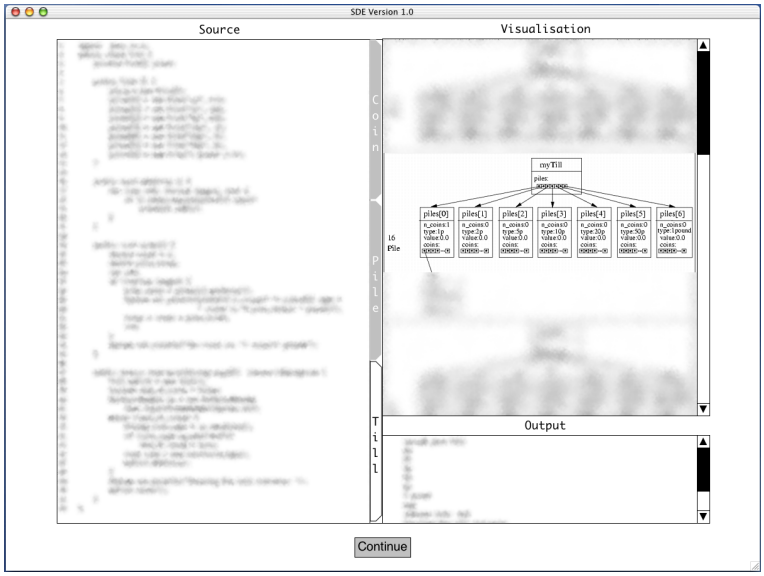

Figure 1. The debugging environment used by participants

multiple, linked, concurrently displayed representations. These include program code listings, data-flow and control-flow visualisations, output displays, etc.

3. Method

The aim of this work was to relate debugging behaviour, especially representation use and co-ordination, to debugging accuracy, representation modality and perspective.

This experiment considered four independent variables, three within subjects and one between subjects, and three dependent variables. The within subjects variables were visualisation modality (textual or graphical), visualisation perspective (data structure or control-flow), and type of error (data structure or control-flow). The between subjects variable was participant's programming experience (less experienced or more experienced), defined by a combination of their Java and general programming experience. The three dependent variables were debugging accuracy, accumulated fixation time and switching frequency between the available representations. Accumulated fixation time refers to the total time participants spent focusing on each representation for each of the debugging sessions. Switching frequency refers to the total number of switches involving each possible pair of representations (code and visualisation, code and output and visualisation and code) for each of the debugging sessions.

3.1. The experimental debugging environment

The Java SDE enabled participants to see the program code, its output for a sample execution, and a visualisation of this execution. A screen shot of the system is shown in Figure 1. Participants were able to see the several program class files in the code window, one at a time, through the use of the side-tabs ('coin', 'pile', 'till' in the example shown). Also, the visualisation window presented a visualisation of the program's execution similar to those found in Object-Oriented software development environments [17]. This visualisation highlighted either data structure or control-flow aspects. These representations were selected because research in Object-Oriented program comprehension has suggested that function and data element information is highlighted in languages of this programming paradigm while control-flow is obscured (see Section 2.3). In our experiments, these representations, and the Java SDE, were static in that participants were presented with selected precomputed information about the program execution. We chose to present information in this way so that we could control for issues like the increased complexity of dealing with a full debugging environment and the ephemeral nature of the information presented by a dynamic debugging tool, which, as mentioned in Section 2.1, could have played a role in the discrepancy of results reported in [12] and [13].

The SDE was implemented on top of a modified version of the Restricted Focus Viewer (RFV) [2]. The SDE presents image stimuli in a blurred form. When the user clicks an image, a section of it around the mouse pointer becomes focused. In this way, the program restricts how

much of a stimulus can be seen clearly and allows visual attention to be tracked as the user moves an unblurred 'foveal' area around the screen. Use of the SDE enabled moment-by-moment representation switching between concurrently displayed, adjacent representations to be captured for later analysis.

A previous study which employed the SDE to validate the suitability of this technology to investigate Java program debugging offered promising results [18]. Specifically, it suggested that debugging performance is not affected by this method of tracking visual attention and that there might be fixation and switching patterns characteristic of superior debugging in this context.

3.2. Participants and procedure

The experimental participants were forty nine computer science undergraduate students from the School of Cognitive and Computing Sciences at Sussex University, U.K. All of the participants had taken a three month introductory course to Java, but their programming experience varied from having only taken this course to a few extra months of Java experience and even having worked as professional programmers. The *less experienced* programmers had on average 3 months of Java experience (basically the duration of the introductory Java course) plus 10.5 months of other programming experience, while the *more experienced* group had on average one year of Java and 13 months of other programming experience.

Participants performed five debugging sessions. The first one was a warm-up session and it employed a functional visualisation. The four main sessions followed, two of them using a data structure and the other two a control-flow visualisation. Also, two of them employed a textual and the other two a graphical visualisation. In this way, the main sessions comprised the four ways in which perspective and modality could be combined, and their order and combinations were counterbalanced across participants and target programs.

The debugging sessions consisted of two phases. In the first phase participants were presented with a specification of the target program. This program specification consisted of two paragraphs describing in plain English the problem that the program was intended to solve, the way it should solve it (detailing the solution steps, specifying which data structures to use and how to handle them), together with some samples of program output (both desired and actual). When participants were clear about the task that the program should solve and also how it should be solved, they moved on to the second phase of the session.

In the second phase participants were presented with three windows containing the program code, a sample interaction with the program and a visualisation which illustrated this interaction. They were allowed up to ten minutes to debug each program. They were instructed to identify as many errors as possible in this program and to report them verbally by stating the class file and line number in which they occurred as well as a brief description of them. They were also encouraged, besides reporting the errors, to think aloud throughout this second phase.

```
  public void add(Coin c) {
    for (int i=0; i<piles.length; i++) {
      if (c.label.equals(piles[i].coin_type))
        piles[0].add(c);
    }
  }

  public static void main(String args[])
                          throws IOException {
    Till myTill = new Till();
    boolean end_of_coins = false;
    BufferedReader in = new BufferedReader
            (new InputStreamReader(System.in));

    while (!end_of_coins) {
      String coin_type = in.readLine();
      if (coin_type.equals("end"))
        end_of_coins = true;

      Coin coin = new Coin(coin_type);
      myTill.add(coin);
    }
    System.out.println("Till contents:");
    myTill.count();
  }
}
```

Figure 2. Segment of the program code for the Till class.

The target programs consisted of five short Java programs. The 'warm-up' session program detects whether a point is inside a rectangle, given the co-ordinates of the point and the vertices of the rectangle. The first and second experimental program print out the names of the children of a sample family. The main difference between these two programs is that the second one is a more sophisticated version of the first one. The third and fourth experimental programs ('Till' programs) count the cash in a cash register till, giving subtotals for the different coin denominations. Again, the main difference between these two versions is that the fourth program is implemented in a more sophisticated way. Some of the code, output for a sample execution session and a control-flow graphical visualisation of this execution for one of the *Till* programs are shown in Figures 2, 3 and 4 respectively.

The programs of the two main debugging sessions were seeded with four errors, and the 'warm-up' session's program was seeded with two errors. The errors of the main

debugging sessions programs can be classified as 'control-flow' and 'data structure'. In this classification, control-flow errors have to do with the execution of the program not following a correct path. For example, the control-flow error in the *Till* program is located in the two last lines of the *while* loop of its *main* procedure. These two lines should be included within an *else* structure, so that the execution of the program either acknowledges an end-of-coins case or adds the new coin to the till, but never follows both paths at the same time.

Data structure errors normally have undesired consequences for the program data structures. For the *Till* program of Figure 2, the data structure error is located within the only instruction of the *if* structure of the *add* method. This error consists of every coin added to the till being sent only to the first money pile, regardless of its type. In this way, the money pile receiving all coins is one which should only accumulate coins of a one-pence denomination.

3.3. Debugging accuracy scoring

The audio recordings of the debugging sessions were analysed to identify the participants' debugging accuracy. Each set of utterances reporting an error was scored according to whether participants identified the place and nature of the error correctly. The place of the error was considered as correct if participants mentioned the line of code where the bug occurred, and only partially correct if they mentioned the method where it happened. Similarly, identifying the nature of the error was considered as correct if participants described it appropriately or if they proposed a correct fix to it. If, for example, they described an effect but not the cause of the error, this second score would be considered as partially correct.

4. Results

The results of the experiment in terms of debugging performance and type of error show that more experienced programmers were able to spot more errors than less experienced ones ($F(1,45) = 5.481$, $p < .01$) and that that control-flow errors were more difficult to spot than data structure errors ($F(1,47) = 9.101$, $p < .01$). Also, there was an interaction effect between level of experience, type of error and modality ($F(1,47) = 4.158$, $p < .05$). This interaction effect is illustrated in Figure 5. This figure shows that when dealing with data structure errors, less experienced programmers found graphical representations more useful, whereas for more experienced programmers modality of visualisation did not have an effect. On the other hand, for control-flow errors, less experienced programmers seem to do better with textual representations, while more experienced found graphical visualisations more useful.

```
rsunx% java Till
5p
1p
2p
5p
1 pound
end
unknown coin: end


Till contents:
5 1p coins is 0.05 pounds
0 2p coins is 0.0 pounds
0 5p coins is 0.0 pounds
0 10p coins is 0.0 pounds
0 20p coins is 0.0 pounds
0 50p coins is 0.0 pounds
0 1 pound coins is 0.0 pounds
The total is: 0.0 pounds

rsunx%
```

Figure 3. Output from a sample execution session of the *Till* program.

It should be noted that there were no significant results linking programming perspective to debugging accuracy, either on their own or with any of the other independent variables.

4.1. Representation use

This part of the analysis tried to relate switching frequency and accumulated fixation time to programming experience, type of error and visualisation perspective and modality.

Two separate ANOVAs were run, one for switching frequency and another for fixation time. The results for switching frequency show a significance effect for switching type ($F(2,46) = 94.527$, $p < .01$), switches between the code and the visualisation being by far the most common, and near significance for the interaction effect between level of experience, type of switch and perspective ($F(2,46) = 3.045$, $p = .057$). This interaction effect is illustrated in Figure 6. This figure shows that more experienced participants exhibit a higher frequency of switching than less experienced ones for switches between the code and output representations, but even more so when using control-flow visualisations.

The results for fixation time show that participants spent the most time looking at the code representation ($F(2,46) = 3459.542$, $p < .01$), and that there was an interaction effect between type of representation and perspective ($F(2,46) = 7.595$, $p < .01$). Figure 7 illustrates this interaction effect graphically. This figure compares the amount of time people spent looking at the different representations for both

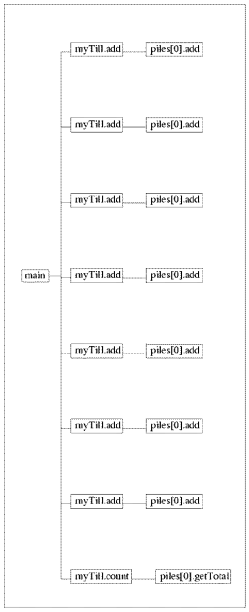

Figure 4. Control-flow graphical visualisation of a sample execution session of the *Till* program.

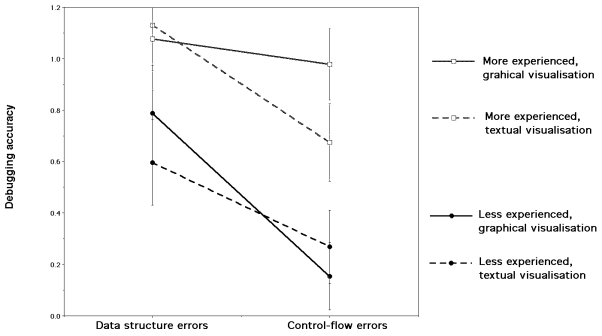

Figure 5. Debugging performance for programming experience, type of error and visualisation modality

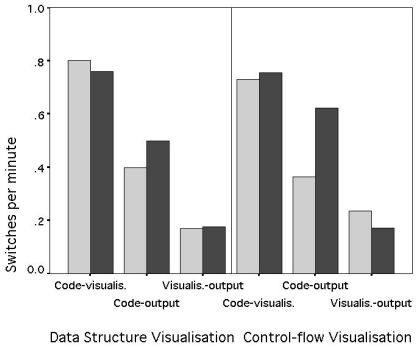

Figure 6. Switching frequency for programming experience,type of switch and visualisation modality. Lighter bars are for the less experienced group, darker for the more experienced.

the data structure and the control-flow visualisation conditions. It can be seen that people spent more time looking at the code in the control-flow condition than in the data structure one. On the other hand, participants spent more time looking at the visualisation in the data structure condition than in the control-flow condition.

5. Discussion

This investigation aimed to relate debugging performance to representation use in a multi-representational, multi-modal debugging environments similar to those found in commercial software development environments and software visualisation packages [17]. These sorts of environment are characterised by having several concurrently displayed representations of the program. There is a central representation, the program code, and a series of secondary representations that support it (program output and execution visualisations). Because software debugging environments are an important tool for novice programmers, modelling the process of representation use in this sort of environment is of central relevance for educational purposes.

The experimental results do not favour a particular type of representation as the best one but suggest an interaction between programming experience, error type and representation modality. They also relate programming experience to specific switching behaviour, and representation fixation times to programming perspective.

It is not surprising that the more experienced programmers were more successful than the less experienced participants in detecting errors in the programs. Also, the fact that control-flow errors were difficult to find is in agreement with research in the Object-Oriented programming comprehension area which suggests that this kind of language highlights function as well as static data element information at the cost of obscuring control-flow information [4, 20]. This result contrasts with debugging studies for languages of other programming paradigms, such as the one in [9], which compared debugging performance for Pascal and BASIC and did not find control-flow errors particularly difficult for participants.

The results suggest an interaction between program-

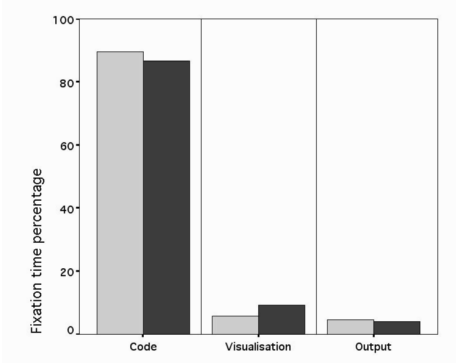

Figure 7. Fixation time for available representation and visualisation perspective. Lighter bars are for the control-flow condition, darker for data structure

ming experience, type of error and representation modality. When dealing with data structure errors, less experienced programmers found graphical representations more useful, whereas for more experienced programmers modality of visualisation did not have an effect. On the other hand, for control-flow errors, less experienced programmers seem to do better with textual presentations, while more experienced participants found graphical visualisations more useful. This seems to suggest that graphical representations are most useful when the debugging task is challenging enough for programmers (data structure errors for the less experienced, control-flow bugs for the more experienced).

Similarly to previous findings [18], the results show that participants spent most of the time focusing on the code window. This is not surprising, because the code is clearly the main representation of the program. It was also mentioned that people spent more time looking at the code in the control-flow condition than in the data structure condition, and that on the other hand, participants spent more time looking at the visualisation in the data structure condition than in the control-flow condition (see Figure 6). This fact can be explained by at least two causes (or a combination of them). The first one is that data structure visualisations were more difficult to understand than the control-flow ones for participants and therefore they had to spend a longer amount of time trying to comprehend them. The second explanation is that data structure information was more useful than control-flow and because of this the data structure visualisation was more important in fixation time terms than the control-flow one. The first explanation seems to be the more likely one due to the fact that the results did not show a better debugging performance for the data structure visualisation condition.

In fact, there were no significant effects involving debugging accuracy and visualisation perspective, which suggests that there were no differences in the effectiveness of data structure versus control-flow visualisations for any type of error. This result is in apparent disagreement with research in the area, because according to [8], a match between type of error and type of visualisation should produce higher levels of accuracy. One factor that might explain this discrepancy is that studies supporting the match-mismatch hypothesis [4, 8, 20] have mainly employed a single representation (the program code), while in this study there were several representations, and the one employed to test this hypothesis (i.e. the visualisation) had, as shown by the fixation times data, only a supportive role in the task.

Another possibility for this lack of support for the match-mismatch hypothesis is the amount of information presented in the visualisation window. Because of the static nature of the experimental debugging tool, the visualisation window presented a sequence of visualisations of the execution state in chronological order (early execution states appeared at the top of this window while those at the end at the bottom of it). The information relevant to discover the errors in the program might have been there, but to search for it might have not been an easy task. So, probably the match-mismatch hypothesis was not supported because issues relating to information decoding and search were more relevant in this case.

The experimental results regarding switching behavior confirmed previous findings [18] that suggest that switches of the unblurred spot between the code and the visualisation are the most common type of switch performed by participants. The difference when considering skill level was that the more experienced performed a higher frequency of switching for the code and output representations, but even more so when using control-flow visualisations. One way to interpret this result is that more experienced programmers exhibit a more balanced switching behaviour between the main and the secondary representations. However, the reason why they performed more switches in the control-flow visualisation condition is not clear. It seems as if they knew that they would not find the information that they were looking for in the visualisation window for the control-flow condition, so they resorted to use the output window instead. This way of reasoning would suggest that data structure visualisations contained more relevant information than the control-flow ones in this case. However, as mentioned above, the results did not show a better debugging performance for the data structure visualisation condition.

6. Conclusions

This study investigated Java program debugging performance and behaviour through the use of a software debugging environment that provided concurrently displayed, ad-

jacent, multiple and linked representations and that allowed visual attention switches of participants to be tracked.

The experimental results suggest that graphical representations might be more useful than textual ones when the degree of difficulty of the debugging task poses a challenge to programmers. Additionally, the results link debugging performance to switching behaviour, suggesting that experienced programmers exhibit a more balanced switching behaviour between the main and the secondary representations. The results also give support to research in the Object-Oriented program comprehension area that suggests that languages of this type highlight static data element information at the cost of obscuring control-flow.

The results of the experiment reported here need to be confirmed by more empirical studies with different experimental settings. One experimental factor that is important to manipulate is the use of a dynamic debugging environment instead of, as in this case, a static one. The use of a dynamic debugging environment might impose an additional cognitive load on participants but will enhance the ecological validity of the experimental task by providing an interactive (and more authentic) SDE environment. This approach will also avoid the need for lengthy sequences and difficult-to-search static representations in the visualisation window, since a single visualisation can be updated in real time.

7. Acknowledgments

This work is supported by the EPSRC grant GR/N64199. The authors would like to thank the participants for taking part in the study.

References

[1] S. Ainsworth, D. Wood, and P. Bibby. Co-ordinating multiple representations in computer based learning environments. In P. Brna, A. Paiva, and J. Self, editors, *Proceedings of the 1996 European Conference on Artificial Intelligence on Education*, pages 336–342, Lisbon, Portugal, 1996.

[2] A. Blackwell, A. Jansen, and K. Marriott. Restricted focus viewer: a tool for tracking visual attention. In M. Anderson, P. Cheng, and V. Haarslev, editors, *Theory and Application of Diagrams. Lecture Notes in Artificial Intelligence 1889*, pages 162–177. Springer-Verlag, 2000.

[3] M. D. Byrne, R. Catrambone, and J. T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33:253–278, 1999.

[4] C. L. Corritore and S. Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human Computer Studies*, 50:61–83, 1999.

[5] T. de Jon, S. Ainsworth, M. Dobson, A. van der Hulst, J. Levonen, and P. Reimann. Acquiring knowledge in science and mathematics: The use of multiple representations in technology-based learning environments. In M. W. van Someren, P. Reimann, H. P. A. Boshuizen, and T. de Jon, editors, *Learning with Multiple Representations*, pages 9–40. Elsevier Science, Oxford, U.K., 1998.

[6] M. Eisenstadt, M. Brayshaw, and J. Paine. *The Transparent Prolog Machine*. Intellect, Oxford, England, 1991.

[7] D. J. Gilmore. Models of debugging. *Acta psychologica*, 78(1):151–172, 1991.

[8] D. J. Gilmore and T. R. G. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21(1):31–48, 1984.

[9] D. J. Gilmore and T. R. G. Green. Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology*, 40A:423–442, 1988.

[10] J. Good. *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, U.K., 1999.

[11] D. C. Merrill, B. J. Reiser, R. Beekelaar, and A. Hamid. Making processes visible: scaffolding learning with reasoning-congruent representations. *Lecture Notes in Computer Science*, 608:103–110, 1992.

[12] P. Mulholland. Using a fine-grained comparative evaluation technique to understand and design software visualization tools. In S. Wiedenbeck and J. Scholtz, editors, *Empirical Studies of Programmers, seventh workshop*, pages 91–108, New York, 1997. ACM press.

[13] M. J. Patel, B. du Boulay, and C. Taylor. Comparison of contrasting Prolog trace output formats. *International Journal of Human Computer Studies*, 47:289–322, 1997.

[14] N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers, second workshop*, pages 100–113, Norwood, New Jersey, 1987. Ablex.

[15] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[16] P. Romero. Focal structures and information types in Prolog. *International Journal of Human Computer Studies*, 54:211–236, 2001.

[17] P. Romero, R. Cox, B. du Boulay, and R. Lutz. A survey of representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing*, in press.

[18] P. Romero, R. Cox, B. du Boulay, and R. Lutz. Visual attention and representation switching during java program debugging: A study using the restricted focus viewer. *Lecture Notes in Artificial Intelligence*, 2317, in press.

[19] I. Vessey. Toward a theory of computer program bugs: an empirical test. *International Journal of Man-Machine Studies*, 30(1):23–46, 1989.

[20] S. Wiedenbeck and V. Ramalingam. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human Computer Studies*, 51:71–87, 1999.

[21] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. L. Corritore. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11:255–282, 1999.